VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Branch and Bound Approach for Scheduling Problem

Branch a Bound v problematice plánování

2015 Bc. Pavel Ivánek

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Diploma Thesis Assignment

Student: **Bc. Pavel Ivánek**

Study Programme: N2647 Information and Communication Technology

Study Branch: 2612T025 Computer Science and Technology

Title: Branch and Bound Approach for Scheduling Problem
Branch a Bound v problematice plánování

The thesis language: English

Description:

Branch and bound (BB) is by far the most widely used tool for solving large scale NP - hard combinatorial optimization problems. BB is, however, an algorithm paradigm, which has to be filled out for each specific problem type, and numerous choices for each of the components exist.
This project will require the student to apply the BB approach to the Flow Shop Scheduling Problem (FSSP) and/or its variants and produce comparable upper and lower bounds.

The detailed tasks include:
1. Adaptation of the NEH algorithm as the generator of the initial solution.
2. Analysis of a local search heuristic for the generation of the solution list.
3. Determining the total size of the feasible solution list.
4. Development of the relaxation rule for the algorithm, which would define how many routes, should be negated.
5. Determination of the termination criteria of the algorithm
6. Adaptation of the algorithm for the scheduling problem

The output of the thesis will be the upper and lower bound estimations of newly constructed problem instances, which will then be tested by evolutionary algorithms.

References:

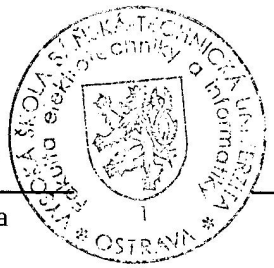
[1] Brusco, M and Stahi S. 2010. Branch-and-Bound Applications in Combinatorial Data Analysis. Springer, Germany
[2] Cehn.X and Bushnell M. 1995. Efficient Branch and Bound Search with Application to Computer-Aided Design. Springer, Germany
[3] Golden, B., Raghavan, S and Wasil E., 2010. The Vehicle Routing Problem: Latest Advances and New Challenges. Springer, Germany
[4] Tooth, P and Vigo D., 2001. The Vehicle Routing Problem (Monographs on Discrete Mathematics and Applications). SIAM, USA

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **doc. MSc. Donald David Davendra, Ph.D.**

Date of issue: 01.09.2013

Date of submission: 07.05.2015



doc. Dr. Ing. Eduard Sojka
Head of Department

prof. RNDr. Václav Snášel, CSc.
Dean of Faculty

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

In Ostrava 7 May 2015

Pavel Ivanek .....

First of all, I would like to thank my Master's Thesis advisors doc. MSc. Donald David Davendra, Ph.D. for his support and guidance. His insight, suggestions and comments made this thesis possible. I would also like to thank my friends who encouraged me and belived in me in difficult times. And, finally, thanks to my family for being close to me.

Abstract

This thesis presents Branch and Bound method for implementing algorithms for solving different kind of discrete combinatorial problems by using systematic enumeration of all solution candidates. Using bounding values, which defines the range where optimal solution is located, we limit potential solution space. The target of this thesis is to implement such an algorithm for a specific scheduling problem and its variations. The resulting implementation is then used to calculate the set of partial results based on well-known scheduling problem instances.

Keywords: branch and bound method, scheduling, makespan, lower and upper bound

Abstrakt

Práce představuje metodu větví a mezí pro vytváření algoritmů pro řešení různých diskrétních kombinatorických problému, kde dochází k systematickému procházení všech potenciálních kandidátů. Při hlédání optimálního řešení se používají mezní hodnoty, které jednoznačně stanovují, v jakém intervalu se řešení vyskytují. Cílem práce je implementovat takový algoritmus na konkrétní plánovací problém a jeho jednotlivé varianty. Výsledná implementace je poté použitá pro vytvoření sady průběžných výsledků pro předem definované instance plánovacího problému.

Klíčová slova: metoda větví a mezí, problematika plánování, objektivní funkce, horní a dolní meze

List of Acronyms

B&B	–	Branch and Bound
IDE	–	Integrated Development Environment
FSBlock	–	Flow shop - blocking
FSNoWait	–	Flow shop - no wait
FSP	–	Flow shop - permutation
FSS	–	Flow shop - simple

Contents

List of Tables

List of Figures

List of Source Code statements

1 Introduction

Planning is the first thing to do when starting any production process. A plan to prepare a sequence of tasks needed to create expected product should be prepared in such a way that it is the most efficient. Behind the term "efficient" we can see many things e.g. no delays, load spread over all resources or even parallelism. Companies are trying to keep all their resource as productive as possible. The need to plan any kind of resources can be found in many various areas from the manufacturing process to software development process.

In most cases it is not an easy task to find a schedule that is *optimal*. It this context optimal means feasible and of a good quality. One particular area where planning is the key part is scheduling. The resulting schedule is qualified based on a certain attribute or collection of attributes, e.g. schedule length. The process of finding an optimal schedule is about improving the schedule in order to get better one.

It can be very hard and may require to examine all possible schedules in order to find the best one. There are situations where it is sufficient to find a "good enough" solution which can be found in a reasonable time.

The methodology to prepare an algorithm that would find some solution soon and then try to improve this solution is called *Branch and Bound*. This paradigm defines key parts of the algorithm and how they are used in order to examine a whole solution space using an enumeration method. Not all potential solutions have to be examined thanks to limits (or bounds) that defines where the optimal solution lies.

Of course each optimization problem can be defined differently so the algorithm needs to be designed specifically for each problem. Branch and bound defines rules that should be followed during the algorithm design.

The outcome of the planning process is a schedule which defines what, where and when should be done. Scheduling contains a lot of different kinds of problems. The problems differ in the size of input instances (amount of tasks and resources), internal conditions (resulting schedule must meet these conditions) or task of resource dependent conditions.

Flow shop scheduling problems are a class of scheduling problems in which the flow control enables sequencing of jobs on a set of machines in a given processing order. The problems place emphasis on completion time(s) where flow of processing tasks is desired with a minimum of idle time and waiting time. Flow shop scheduling is a special type of job shop scheduling in which jobs are assigned to resources at particular times.

The enumeration procedure may, in the worst case scenario, lead to examining all feasible solutions which could be very time consuming. By using the B&B algorithm a partial solution with certain boundaries can be produced and that can be used for further processing.

In this thesis it is not expected to find an optimal solution for all problem instances prepared for experimentation but to give partial solutions with their bounds. In order to get even partial solution an algorithm needs to be prepared. So thesis also contains description of the used branch and bound algorithm together with functions used for calculating boundaries needed for feasible solution space reduction. To be able to better

understand the algorithm, example calculation will also be presented. The implentation is then used for experimentation on benchmark datasets and in the end, results are presented.

2 Branch and Bound

A large number of real-world problems requires planning of smaller tasks which together in specific order represents a schedule. Better order of these tasks can of course lead to better efficiency which is exactly what we are looking for. In reality, the problem can be to organize the separate tasks in the manufacturing process, planning complex schedules, identifying the shortest paths in logistics and many more.

These planning problems are called combinatorial optimization problems which are very often $\mathcal{NP}$-hard. For such problems there is no known polynomial algorithm, and so more complex algorithms are needed. Solving $\mathcal{NP}$-hard discrete optimization problems is extremely difficult and require very efficient algorithms and the B&B paradigm tells us the approach on constructing such algorithms.

First appearance of B&B was in 1960 [1] in context of integer programming. The first problem to be adopted B&B was "Traveling Salesman Problem" where the actual term "Branch and Bound" was mentioned for the first time. Currently the B&B is a widely used tool for solving large scale $\mathcal{NP}$-hard combinatorial optimization problems. B&B itself is not an algorithm but it describes a way to prepare an algorithm. Each problem requires specific algorithm based on B&B methodology.

2.1 General description and terminology

Branch and Bound algorithm design paradigm as such describes a way how to prepare algorithms for solving discrete and combinatorial optimization problems using systematic enumeration of candidate solutions. The algorithm searches the complete solution space in order to find the best solution. During the process potential solutions are being calculated and thus the solution is being improved on the way. Processing time can of course be limited by specifying a condition which defines a "good enough" solution.

During the solving process, the status of the solution is described by a best solution found so far and the list of yet unexplored subsets of potential solutions. Initially only one subset exists, the complete solution space and the best solution so far is defined as a ∞. Best solution found so far is usually called *incumbent*.

Initially complete solution space is divided into dynamically generated subspaces and searched through separately by the algorithm. Such a division is referred to as *branching*. By dividing the solution spaces into smaller and smaller portions, the whole structure can be visualized as a dynamically generated tree (see Figure 1) where complete solution space represents a root node and subsets represent child nodes and leaves if the subset is not or cannot be divided anymore. Each iteration in classic B&B algorithm processes one such node.

Each node in the tree needs to be evaluated. Evaluation is done by using *bounding functions* which calculate two values, *lower bound* and *upper bound*. Calculated values represent the range in between all possible solution found in this branch are situated. The bounding function simply tells us whether that subspace can contain a better solution than the current best solution. If not, then the whole subspace can be discarded. The process where non-promising branches are eliminated is called *pruning*.

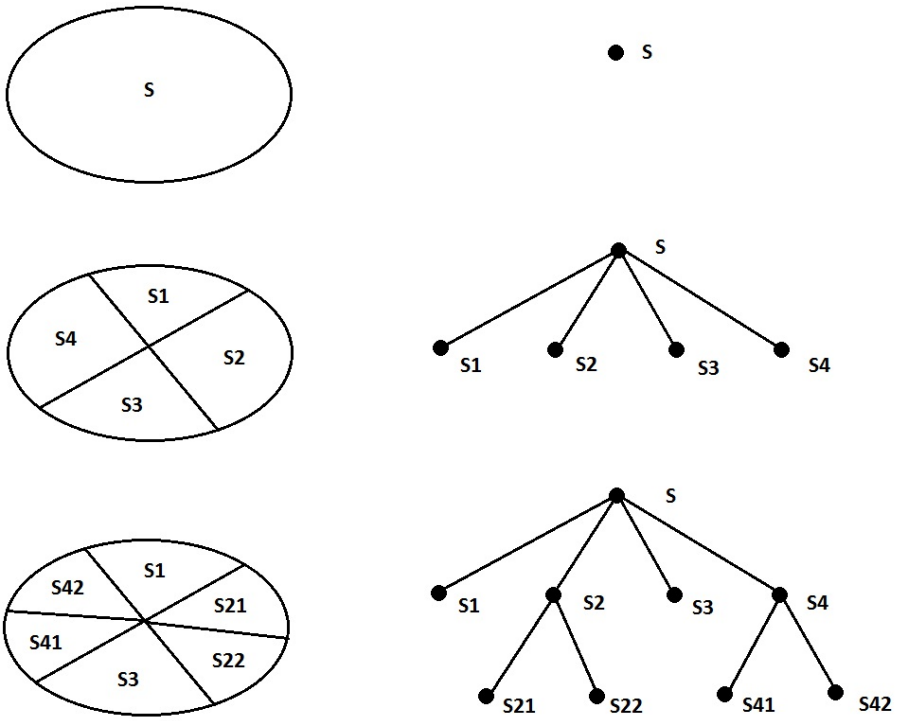


Figure 1: Solution space tree

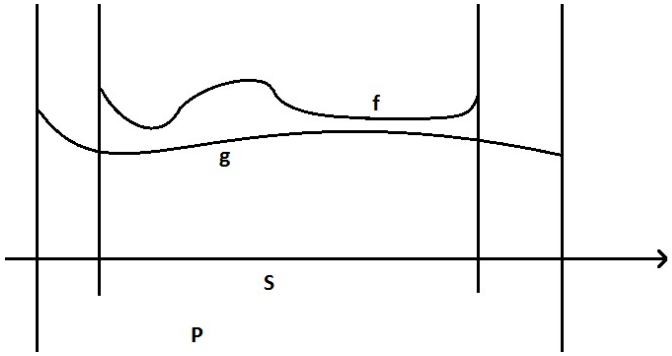


Figure 2: The relation between objective function f and bounding function g on the sets of potential P and feasible S solutions of a problem.

In case the subspace contains only a single solution, it is compared to the incumbent keeping the best of it. All other branches that were not eliminated are stored in the pool of *live nodes* together with their bounds. Having the set of all live nodes, there is a need to decide which branch will be processed next. *Strategy for selecting next subsection* can dramatically influence the efficiency of the algorithm. There exist several strategies which will be described in the text later.

2.2 Algorithm parts

I the following I consider minimization problems (maximization problems can be dealt with similarly). In minimization problems we are trying to minimize a function $f(x)$ of variables $(x_1, x_2, \ldots, x_n)$ over a region of *feasible solutions*, S :

$$min_{x \in S} f(x)$$

The function f is called *objective function* and may be of any type and the best solution can be found by simple calculating $f(x), \forall x \in S$. This approach is called brute force algorithm and requires enormous amount of time because function f is *expensive* (difficult to calculate).

In many cases, a set of *potential solutions*, P, where $S \subseteq P$, for which f is still well defined, bounding function $g(x)$ defined on S (or P) with property that $g(x) \leq f(x), \forall x \in S$ (resp. P) arises naturally [2]. Bounding function g is used to find the tightest limit for objective function f because its calculation is *cheap*.

As described before, a B&B algorithm for a minimization problem consists of three main parts:

1. *a bounding function* providing lower bound for the best solution value that can be found in the subspace,

2. *a strategy for selecting* next subspace to be processed by the algorithm,

3. *a branching rule* which defines a way to split currently processed subspace into even smaller subsets which will be processed in the next iteration of the algorithm.

Each iteration of B&B algorithm requires a node to be selected for exploration from the pool of live nodes. The live nodes in this case represents subsets of feasible solutions. In general, there exist two strategies for selection of the next node. If the *eager* strategy is used, a branching is performed. Two or more new subsets are generated and their bounds are calculated. If the lower bound of the child node is better than the current best solution, it is kept. If the lower bound indicates that this child node cannot contain better solution, it is discarded (or *fathomed*), otherwise it is added to the pool of live nodes. These comparisons are done for all newly generated child nodes.

Eager Branch and Bound [2]

Incumbent := ∞; LB(P_0) := $g(P_0)$; Live := {($P_0, LB(P_0)$)}

repeat **until** Live = $\emptyset$
Select the node P from Live to be processed; Live := Live $\setminus$ {P}
Branch on P generating $P_1, ...P_k$;
for 1 $\leq i \leq k$ **do**
  Bound P_i : LB(P_i) := $g(P_i)$;
    If LB(P_i) = $f(X)$ **for** a feasible solution X
      and $f(X) < Incumbent$ then
        Incumbent := $f(X)$; Solution := X;
        go to EndBound; \\
    If LB(P_i) $\geq$ Incumbent then fathom P_i
      else Live := Live $\cup\{(P_i, LB(P_i))\}$
EndBound;

OptimalSolution := Solution; OptimumValue := Incumbent

Statement 1: Eager B&B algorithm

The other selection strategy is called *lazy*. The order of calculation of bounds and branching is reversed. Firstly, the bounds are calculated for the father node before children are generated thus starting the whole process by calculating the bounds for the root node. It means that the initial value does not need to be calculated in initialization part but is simply the first step in the iteration.

Lazy Branch and Bound [2]

```
Incumbent := −∞; Live := {(P_0, −∞))}

repeat until Live = ∅
Select the node P from Live to be processed; Live := Live \ {P}
Bound P : LB(P) := g(P);
   if LB(P) = f(X) for a feasible solution X
      and f(X) < Incumbent then
         Incumbent := f(X); Solution := X;
         go to EndBound;
   if LB(P) ≥ Incumbent then fathom P
      Branch on P generating P_1, ...P_k;
         for 1 ≤ i ≤ k do
            Live := Live ∪{(P_i, LB(P))} \\
EndBound;

OptimalSolution := Solution; OptimumValue := Incumbent
```

Statement 2: Lazy B&B algorithm

2.2.1 Bounding function

The key component of Branch and Bound algorithm is a bounding function. Boundaries calculated for a given potential solution space are telling us the range of solutions this space contains. The better (more accurate) results this function is giving the more quality it is. Of course with quality comes the complexity and computational demands. Ideally the value of a bounding function equals the best feasible solution to the problem.

It is always a trade between quality of the bounding function and its complexity. In reality, since we are looking for a solution of a $\mathcal{NP}$-hard problem, bounds must be calculated using only limited amount of computational time (i.e. in polynomial time).

In case the bounding function is giving bounds that are close to the optimal values, it is called *strong*. If it produces results far from the optimal solution, it is called *weak*.

2.2.2 Strategy for selecting live solution subspace

List of live nodes which represents the possibility to choose the next best fitting potential solution subspace which would ideally contain the best solution. If you imagine a tree of potential solution subspaces, each level smaller, potential solutions are represented as leaves of this tree, we are trying to find a path to the best solution throughout the whole tree. Of course the shorter the path the better, i. e. the less potential solution space we have to examine the better. In theory, we must have a strategy how to choose the next subproblem.

The strategy for selecting next live subproblem to investigate usually reflects a trade between keeping the number of unexplored nodes in the search tree low and staying within the memory capacity of the computer used.

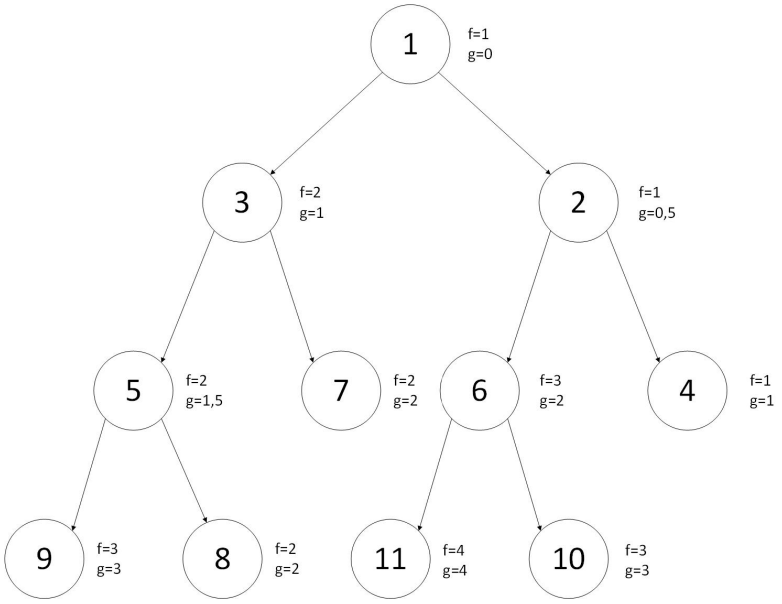


Figure 3: Best first search strategy

If the algorithm always selects among the live nodes the one with the lowest bound, the strategy is called *best first search* (Figure 3) . This strategy does not necessarily provide a quick result. It is more likely that the feasible solutions are deep in the tree structure.

Depth first search strategy chooses the deepest node to be processed next in order to get at least some result very quickly (Figure 4). The risk of this strategy is that if a lower bound is not very accurate then it can happen that many incorrect nodes are processed.

On the other hand the *breath first search* strategy examines all nodes of the same depth in the tree before it goes deeper the tree (Figure 5).

2.2.3 Branching rule

In context of Branch and Bound, branching is a subdivision of a part of the search space. If the subspace is subdivided into two pieces, the branching is called *dichotomic*, otherwise it is called *polytomic* branching.

The fact that the algorithm is finite is ensured if the subdivided solution space is smaller than the original solution space and the number of feasible solutions for the original problem is finite. Usually branching produces solution subspaces that are disjoint. In this case the feasible solution can appear only in one subspace.

2.3 Example

As an example, let's take an instance of a job shop scheduling problem. The job shop problem is an optimization problem in which we are looking for a schedule to assign ideal jobs to resources at particular times. There exist multiple versions of this using

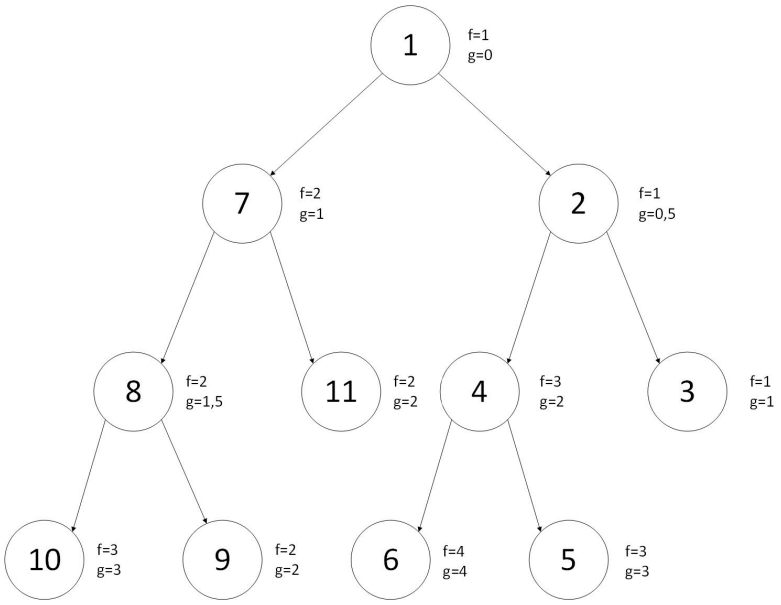


Figure 4: Depth first search strategy

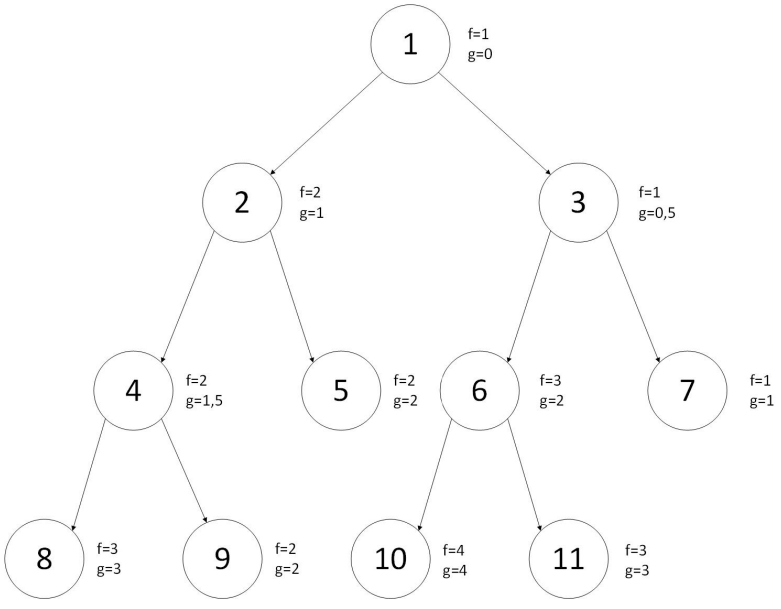


Figure 5: Breath first search strategy

Jobs	1	2	3	4
p_j	12	8	15	9
d_j	16	26	25	27

Table 1: B&B example assignment

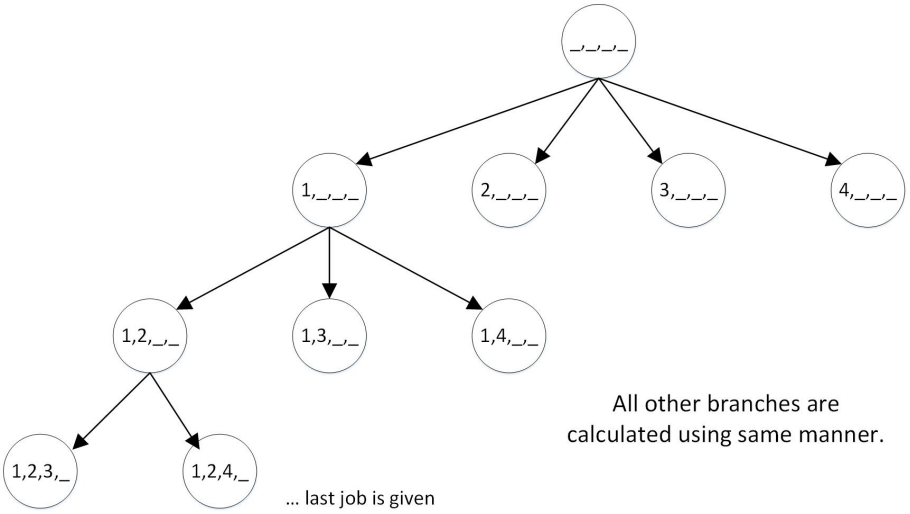


Figure 6: Example tree structure

different amount of resources and different conditions on the schedule itself. For further study of scheduling problems I would recommend [3].

For the purpose of an example, we take the simplest version of this problem $1||\sum T_j$ (assign jobs to a single machine in a way to minimize the processing time). It is very unlikely that there can be developed an algorithm that would find the solution for this problem in polynomial time because this problem is $\mathcal{NP}$-hard.

Input data are represented by the following table:

Processing time p_j of a job j tells us how long it takes to process the task. Dute date d_j of job j states when a task can be completed at the latest.

A solution is a schedule representing an assignment of the jobs to the particular positions in the job sequence running on a single machine.

The idea behind the algorithm is to start with an empty set of assigned jobs. During the processing a job is chosen and added to the set representing scheduling sequence on a particular position. Of course we will begin at first position and continue until all jobs are assigned. This procedure will generate the whole branching tree visualized in Figure 6.

In order to avoid checking all possible permutation, we will use a bounding function which will evaluate each partial schedule. Consider a partial schedule at step k after k jobs have been assigned to the first k positions, see Figure 7

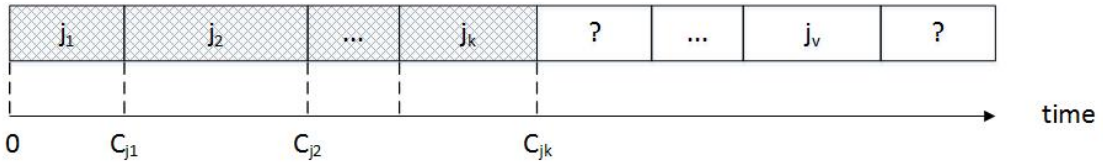


Figure 7: Example time line

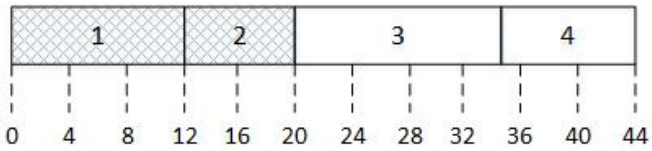


Figure 8: Example schedule - time line

Actual *tardiness of the job assigned to position* k is given as

$$T_{j_k} = max\{C_{j_k} - d_{j_k}, 0\},$$

where d_{j_k} is a due date of job j_k.

Tardiness of any unscheduled job j_v is given as

$$T_{j_v} \geq max\{C_{j_v} - d, 0\},$$

where $d = max\{d_{j_{k+1}}, d_{j_{k+2}}, \cdots, d_{j_n}\}$ maximum due date among unscheduled jobs.

The resulted *bound for a partial schedule where* k *jobs are assigned* is calculated as

$$LB = (T_{j_1} + T_{j_1} + \cdots + T_{j_k}) + (T_{j_{k+1}} + T_{j_{k+2}} + \cdots + T_{j_n}).$$

Calculating Lower Bounds

The lower bound for a partial schedule consists of two parts, scheduled jobs and remaining jobs. In the partial schedule $(1, 2, \_, \_)$ (see Figure 8), tardiness of the first two scheduled jobs is $T_1 + T_2 = max\{C_1 - d_1, 0\} + max\{C_2 - d_2, 0\} = max\{12 - 16, 0\} + max\{20 - 26, 0\} = 0 + 0 = 0$.

Tardiness of the two remaining jobs $3, 4$ is calculated with respect to the large common due date $d = max\{d_3, d_4\} = 27$. $T_3 + T_4 \geq max\{C_3 - d, 0\} + max\{C_4 - d, 0\} = max\{35 - 27, 0\} + max\{44 - 27, 0\} = 8 + 17 = 25$.

It gives us the lower bound for partial schedule $(1, 2, \_, \_)$ is $T_1 + T_2 + T_3 + T_4 = 0 + 25$.

The complete search tree after the optimal solution is found can be seen in Figure 9.

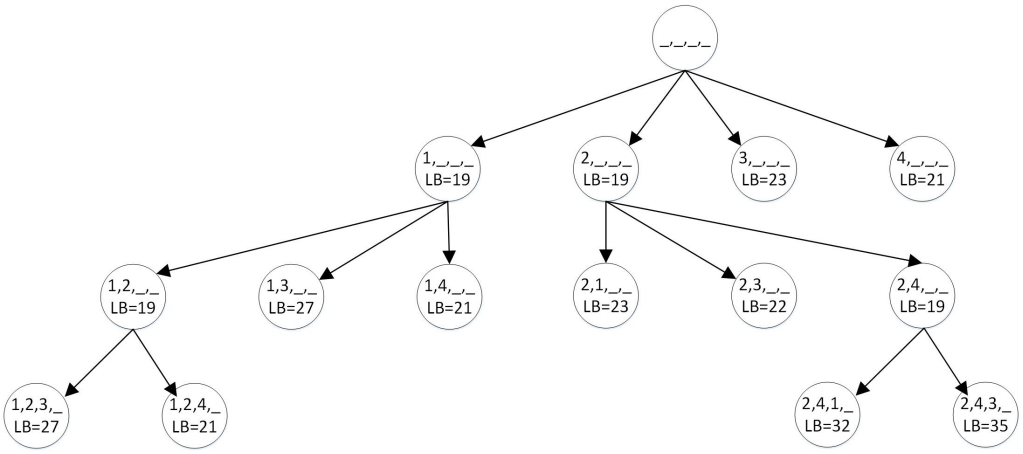


Figure 9: Complete search tree with calculated lower bounds

2.4 Advantages and disadvantages

Advantages

- The algorithm finds optimal solution (can be limited by time or size and can be done in reasonable time).
- Creates relative small solution space (represented by lower and upper bound).

Disadvantages

- Extremely time-consuming (number of nodes can grow very much, additional overhead because of calculation of intermediate results).
- The algorithm must be problem specific (limited reusability).
- Memory consumption can be large (solution space tree can grow very much and information must be stored somewhere).

The algorithm finds some result first and then tries to improve it. Branching and choosing next node strategy can lead to a very big amount of branches and thus calculating many things without actual improvement.

3 Flow Shop

The manufacturing process consists of different variety of operations that need to be done in order to create a final product. The amount of these operations can be quite big and there can be complicated relations between them e.g. one comes after another one. To be able to define the production process precisely, operations need to be divided into relatively small parts which are then suitable for planning the whole processing chain - *jobs*.

Each job usually goes through several stages of processing on a series of locations. Each stage can consist of a number of productive facilities collectively referred to as *machines*. Jobs are defined as set of *tasks* and each of which is performed on a specific machine. The manufacturing process usually has defined constraints e.g. time interval in which the task must be performed, which tasks will come after, tasks that must be completed before the execution can start and many others. These constraints influence the timing of the job processing very much. A *schedule* defines when each task of a given job will be processed and on which machine. When referring to a schedule we most likely mean *feasible* schedule; schedule that satisfies all the constraints.

Job shop scheduling problem, as mentioned earlier in the section about Branch and Bound, is about finding the *optimal schedule* (a feasible schedule) that best achieves a given objective. The objective is usually represented as an objective function. The goal is usually to minimize this function over all feasible schedules.

In general scheduling problems are difficult to solve because the problem may be very large and complex. It is also possible that the problem has multiple possible solutions. The procedure to identify the optimal schedule is slow and exacting.

It is possible to apply *heuristic algorithm* that produces a "good enough" solution. Such a solution can be optimal but there is no guarantee that it really is. Anyway, it can lead to pretty good performance and in most cases it is acceptable.

Definition 3.1 *A **flow shop** is a processing system in which the tasks sequence of each job is fully specified, all jobs visit all machines in the same order and a job never revisit any machine.*

The order of jobs in schedule is called *chain precedence* (see Figure 10) where each node represents a job visiting a machine. The order is defined and no skipping is allowed.

3.1 Overview

Flow shop scheduling problem exists in many variations. The very basic model is *simple flow shop*. Other variations are then derived from this model.

Simple flow shop must meet the following criteria:

- Each machine m can handle only one task at a time.
- Each task of a job requires to be processed on a different machine (no job can visit a machine more than once) and order of processing on machines is the same for all jobs.

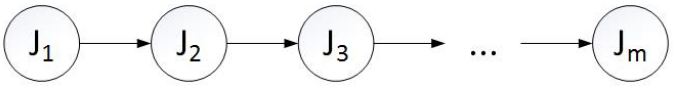


Figure 10: Chain precedence for a job J on $1, ..., m$ machines

- All jobs are availablein in the beginning and remain available without interruption until they are processed.
- Each job, when completed on one machine, is immediately available for next machine (setup time, transfer lags etc. are not allowed).
- Each job can be processed only on one machine at a time (no tasks of one job can be processed simultaneously).
- All n jobs are independent and have to be processed with known requirements: task i of job j requires machine i for a processing time $p_{ij} \geq 0$.
- Once started tasks must be processed to completion (no preemption is allowed).
- Intermediate storage is unlimited (the amount of jobs waiting in the queue before processing is unlimited).

As opposed to *general job shop* scheduling problem definition, the simple flow shop has a condition that *the required machine sequence of different jobs cannot differ*.

Very specific type of flow shop is called *hybrid flow shop* where machines can process several different tasks in parallel. This is sometimes also referred to as *flexible* or *compound* flow shop. This one and many other variations of flow shop problem definitions can be found in [4].

3.2 Terminology

This chapter defines a notation and terminology that will be used throughout the rest of this thesis and is used in many other papers and articles. We will define only notation that will be used in following text therefore you can find also other parameters and terms related to scheduling problem specifications that are not in the following list.

We will start with the parameters. The definition of a particular scheduling problems may contain:

- n: number of jobs to be scheduled.
- m: number of machines in the job.
- J_j: job j where $j \in 1, ..., n$. (Arbitrary order is used for indexing unless otherwise specified.)
- M_i: job i where $i \in 1, ..., m$.

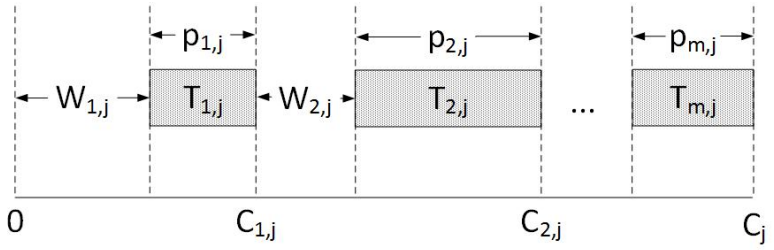


Figure 11: Gantt chart for J_j

- T_{ij}: task i of job J_j. The task to be processed on machine M_i.
- p_{ij}: processing time of T_{ij}.

In the next part you will see a notation related to the schedule itself.

- S: schedule or sequence of jobs.
- S_i: schedule on a machine M_i.
- $C_{ij}(S)$: completion time of T_{ij}.
- $C_i(S)$: completion time of all tasks scheduled on machine M_i.
- $W_{ij}(S)$: waiting time for job J_j between completing task $T_{(i-1)j}$ and T_{ij}.

In order to keep the notation simple, the dependence on schedule S will not be shown when it is obvious from context. So we will use C_i, not $C_i(S)$.

Visualization of a schedule is done using *Gantt charts* that were originally used for project scheduling but can practically visualize any schedule. Each task is represented by a rectangle laying on the horizontal time axis. Length is showing the duration of the task and its location shows the time position as indicated in Figure 11. Each line represents one machine in the schedule.

3.3 Problem classification

Because of the wide variety of the flow shop scheduling problem specifications, classification system was prepared. We will use the classification presented in [5]. Problem is classified using the three part notation $\alpha|\beta|\gamma$.

1. α-field indicates the type of the problem and its size. We will be working with only one type - flow shop. Therefore *Fm*, where m is a number of machines.
2. β contains a list of special features. In our case we will use only three special features which define the difference from simple flow shop problem, or empty in case of simple flow shop.

	J_1	J_2
M_1	1	5
M_2	5	1
M_3	5	1
M_4	1	5

Table 2: Permutation flow shop - Example assignment

- *perm* refers to a permutation version of flow shop
- *(perm)* choosing only permutation schedules even if not required
- *nwt* meaning no-wait flow shop
- *block* indicates that jobs can be blocked in their execution

3. γ holds the criterion that is to be minimized (maximization appears very rarely in scheduling problems and can be transformed to minimization by simply sign reversal). We will be using only one objective function where the objective is to minimize the completion time of the tasks that is the time when the last tasks of each job are completed. C_{max}: latest completion of any job is called the *makespan* and it is used in the most cases.

Simple flow shop according to previous classification is noted as $Fm||C_{max}$ where scheduling n jobs on m machines. Please notice that middle field is empty which means that default assumptions are used.

The complete classification schema with more detailed description and other variations of flow shop scheduling problem can be found in [4].

3.4 Permutation flow shop

The resulting schedule in simple flow shop does not have any requirement on job sequencing on machines as opposed to *permutation flow shop* which adds a requirement to preserve the same order on all machines. In real life manufacturing process, the requirement to follow the same job sequence on all machines is reasonable because frequent skipping among jobs can be inefficient and does not provide good possibilities related to pipeline product processing.

Of course it is possible that the permutation schedule can be optimal also for simple flow shop scheduling problem but we cannot always expect that. Let's see an example.

The following example of scheduling problems comes from [4] and shows the difference between the simple permutation flow shop very evidently. The problem is to find an optimal schedule for 2 jobs running on 4 machines where minimizing the makespan objective: $F4||C_{max}$.
The processing times for all tasks are:

Gantt charts in Figure 12 are showing possible schedules. Notice that in illustration *a)* first task on M_2 takes 5 time units to process therefore there is a time delay of 4 time

units between completion of the second task on M_2 and an initiation of second task on M_3. On the other hand you can see in illustration *b)* that machines M_3 and M_4 are idle for a specific amount of time because they are waiting for the preceding task of the same job to be completed on a previous machine.

Last schedule (illustration *c)*) shows the optimal solution but the job order is not the same on all machines.

In case we are looking for permutation schedule only, it greatly simplifies the calculation of an optimal schedule since there are only $n!$ possibilities how to order jobs. But in case of simple flow shop there are $(n!)^m$ feasible schedules.

In special cases it was proven that permutation schedule will always be optimal over all schedules. It is worth mentioning two situations like that.

Theorem 3.1 *For $Fm||Any$, there exists an optimal schedule with the same job order on the first two machines [6].*

Objective criterion Any represents any function of job completion times.

Theorem 3.2 *For $Fm||C_{max}$, there exists an optimal schedule with the same job order on the last two machines [7].*

These two theorems together say that in two-machine flow shops we need to consider only permutation schedules and in three-machine flow shops we need to consider only permutation schedules but just in case the objective function is makespan.

3.5 No-wait Flow Shop

In manufacturing process there are situation where no delays are allowed during a processing of one job. For example in steel manufacture the two main stages are heating and forming. Hot steel is left in a mould for a specific time. This 3 stage job must not contain any delay between stages.

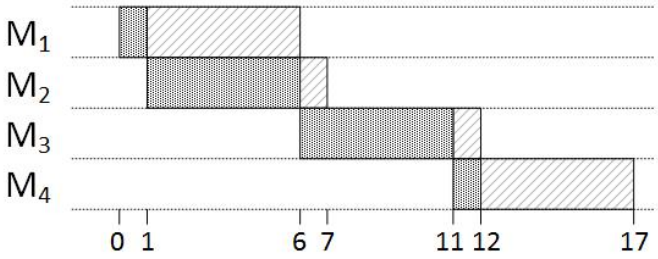
Flow shop scheduling problem with a condition where no such delays are allowed is called *no-wait flow shop* and noted $Fm|nwt, (perm)|C_{max}$.

Definition 3.2 *A job in a no-wait flow shop, once started, must pass through all machines to completion without any delay.*

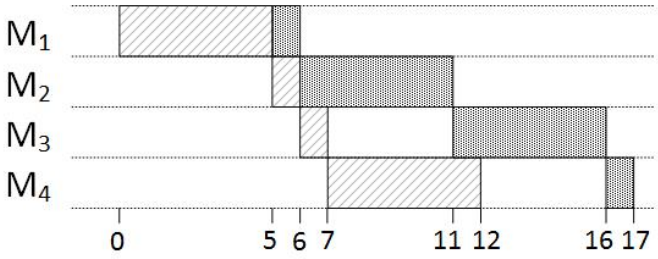
Resulted schedule can contain delays but only between jobs, not between tasks of one job (see Figure 13).

The assumption that only permutation schedules are possible is not generally true. It is only if we add *no-skip* requirement that eliminates the possibility a task can be empty (zero processing time). This requirement is considered valid throughout all the flow shop problem definition but this is the first situation where it has to be taken into consideration.

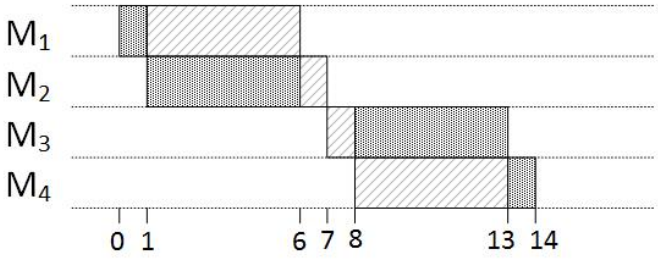
Definition 3.3 *No-skip requirement states that each job must visit each machine [4].*



a) Permutation schedule



b) Other permutation schedule



c) Optimal schedule

Figure 12: Schedules for $F4||C_{max}$

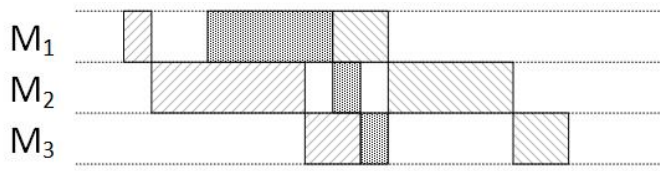


Figure 13: No-wait flow shop example

3.6 Blocking Flow Shop

Optimal schedule is trying to have all the machines working. Often tasks are kept waiting in the queue until machines complete a preceding task but what if these tasks, for some reason, cannot stay there. Waiting tasks need to be stored in a buffer which might not be available or may have only limited size.

Blocking phenomenon most commonly occurs when there are zero buffers and there is simply no place where to keep waiting tasks.

Definition 3.4 *Job that has completed its processing on a given machine cannot leave the machine if the preceding job has not yet completed its processing on the next machine [3].*

The blocked job prevents the next job from starting its processing on the machine.

In the subsequent chapters we assume that jobs can be scheduled only in one and the same order on all machines. Flow shop problem with blocking condition is noted as $Fm|block, (perm)|C_{max}$.

Schedules with blocking are different from those with no waiting condition. However, there is one exception in case there are exactly two machines.

Theorem 3.3 $F2|block, (perm)|$*Any is equivalent to* $F2|nwt, (perm)|Any$.

The proof of this theorem can be found in [4] but we will rather examine an example. In Figure 14 you can see that the only difference between these two schedules is that 3^{rd} and 4^{th} task on M_1 are processed sooner because of the absence of no waiting condition. However this does not provide any improvement makespan because their following tasks on M_2 are not completed soon enough so tasks must wait on M_1 in blocking state.

With higher number of machines ($m > 2$) the results for these two scheduling problems are different.

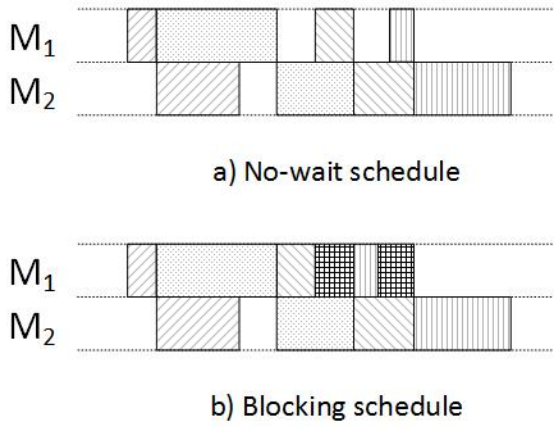


Figure 14: Comparison between $F2|block, (perm)|Any$ and $F2|nwt, (perm)|Any$

4 Implementations

Flow shop scheduling problem is very computationally difficult because of the large amount of jobs (tasks) that must be scheduled. Therefore all the algorithm implementations that will be presented in following section are build up on B&B methodology. Implementation was done in C++ programming language in NetBeans 8.0.2 IDE. Source codes are available in the attachment.

As described earlier each B&B algorithm consists of several parts. Each algorithm is adjusted to solve the particular problem but of course you can find resemblances because each variation of flow shop problem has its foundation in the Simple Flow Shop.

4.1 Simple Flow Shop

Basic variation of flow shop scheduling problem is $Fm||C_{max}$ or simple flow shop. Problem definition (in chapter 3.1) does not contain any additional condition on schedule that would limit the amount of all feasible solution. In case we would be searching for the best solution, it will require to examine every single feasible solution which is $(n!)^m$, where there are $n!$ permutation of job order used on m machines.

It is obvious that even for small number of jobs, the total number of feasible solutions is enormous. B&B gives us possibility not to check every single feasible solution by setting lower and upper bounds. Naturally, the better bounding functions we use the less amount of potential solution subspaces we have to examine.

4.1.1 Lower bound

Because of the generality of the simple flow shop problem definition, there is not much of a basis this function can be built on. However, it is obvious that no job can finish sooner than its total processing time. Even more importantly, entire set of jobs to be scheduled cannot finish sooner then the processing time of the longest job in the schedule. It gives us that we can use the following as a lower bound:

$$LB = \max_{1 \leq i \leq m} \left\{ \sum_{j=1}^{n} p_{ij} \right\}.$$

Presented lower bound function is based on calculation of processing time of tasks executed on a single machine. On the other hand it is easy to calculate and provide quick results.

4.1.2 Makespan

The ultimate goal is to find a schedule with a minimal total completion time C_{max} (or makespan). The procedure to calculate it needs to be able to calculate also a completion time of a partial solution (less than n jobs or less then m machines in the schedule). This feature will be used later in the algorithm where evaluation of even incomplete schedules is needed during the selection of next candidate solution subspace that will be examined.

M_1	T_{11}	T_{13}	T_{14}	T_{12}
M_2	T_{22}	T_{23}	T_{24}	T_{21}
M_3	T_{32}	T_{31}	T_{33}	T_{34}
M_4	T_{44}	T_{43}	T_{41}	T_{42}

Table 3: Simple flow shop - Example assignment for makespan calculation

As you can see in the example schedule in the Table 3 the job order can differ from machine to machine. To simplify the notation in the next paragraph we will use i, j as indexes to a particular tasks. Without any restriction we can use T_{ij} as reference to a task situated on i^{th} machine and on the j^{th} position in the job sequence.

The calculation itself involves 3 steps:

- First machine - all jobs start on the first machine therefore, the completion time of all tasks processed on first machine is

$$C_{1j} = \sum_{k=1}^{j} p_{1k},$$

  where $j \in 1, \ldots, n$.

- First job - all tasks of the first job are starting immediately one after another which gives us

$$T_{i1} = C_{(i-1)1} + p_{i1},$$

  where $i \in 2, \ldots, m$.

- Rest of the schedule - all other tasks follow either the task of the same job on the previous machine or the preceding task of previous job on the same machine depending on whichever ends last

$$C_{ij} = max(C_{(i-1)j}, C_{i(j-1)}) + p_{ij},$$

  where $i \in 2, \ldots, m$ and $j \in 2, \ldots, n$.

The resulting value from calculating the makespan on a complete schedule gives us also the upper bound which represents the best solution found during the time of the execution of B&B algorithm. The upper bound must be stored separately and will be updated every time a better solution (schedule with lower makespan value) is found.

Not all incomplete schedules can be used as an input for makespan calculation procedure. Schedule must always fulfil a condition where a number of assigned jobs to all machines in the schedule is the same.

For example schedule in Table 4 is invalid for makespan calculation in the presented manner.

M_1	T_{11}	T_{12}	T_{13}
M_2	T_{21}	T_{22}	T_{23}
M_3	T_{31}	$\emptyset$	$\emptyset$

Table 4: Simple flow shop - Example of invalid schedule

M_1	J_{11}	J_{12}	...	J_{1n}
M_2	J_{21}	J_{22}	...	J_{2n}
...	...	...	...	...
M_m	J_{m1}	J_{m2}	...	J_{mn}

Table 5: Simple flow shop - Example job indexes

4.1.3 Algorithm

B&B algorithm is an enumeration method that can, in the worst case scenario, end up in examining all feasible solutions. Initially, algorithm starts with an empty schedule and throughout the processing valid schedules are being build up and examined using bounding functions thus building the whole solution space tree.

4.1.3.1 Initialization consists of preparing the initial *Schedule* which is based on a simple arithmetic progression shown in Table 5.

From the initial schedule naturally arises the *Number of jobs* and *Number of machines* in the schedule which of course tells when the schedule is complete.

In each iteration of the algorithm we are working with an incomplete schedule which is gradually being build up. In the beginning, this *Calculated schedule* is empty and represents the root node in the solution space tree. The position in the solution tree is indicated by *Current job and machine* which says at what level and branch we are currently located within the calculations (how many jobs are assigned to how many machines). In case all jobs are assigned to all machines, we have prepared a complete schedule whch cannot be examined futher.

To be able to construct a feasible schedule there is a need to keep track of *Available jobs* because each job must have exactly one task to be processed on each machine in the schedule. This set initially contains all jobs because we are beginning with the empty schedule.

To get a feasible solution quickly a depth first search strategy is chosen. The feasible solution will then be refined during the execution. By using this strategy we incur the risk of getting stuck inside one branch of the solution space tree and it might be difficult to examine an unrelated branch because of potentially big amount of possible feasible solution in the branch.

Current best solution found in every step of algorithm will be stored in *Optimal schedule* structure which is of course initially empty. All feasible solutions that can lead to better schedule are always inside an interval given by *Upper bound* and *Lower bound* val-

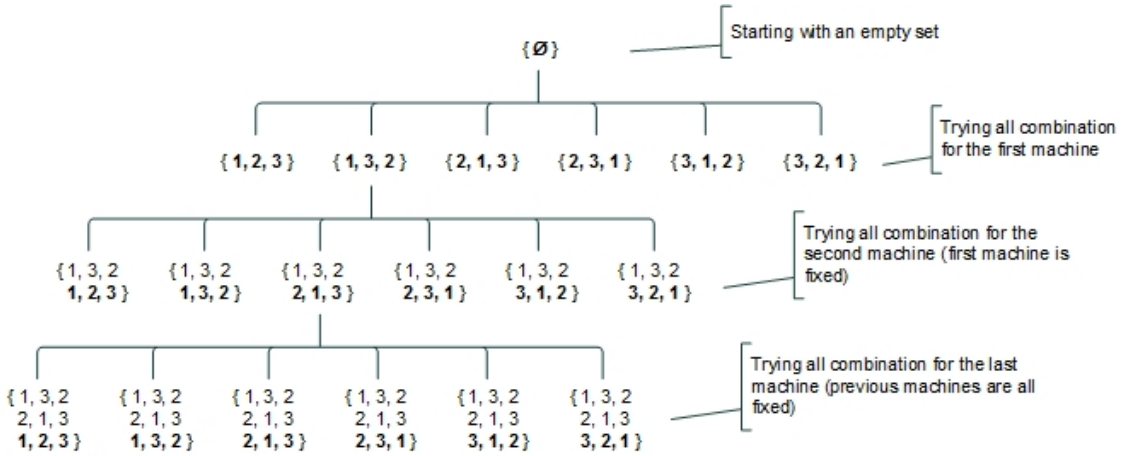


Figure 15: Simple Flow Shop - Solution space tree example

ues. The upper bound value begins as a theoretical maximum and lower bound as a zero (the duration of the schedule cannot be less then zero).

4.1.3.2 Main idea behind the B&B algorithm for simple flow shop problem is to start with an empty schedule and generate all combinations of job order for the first machine. Evaluate each incomplete schedule like that using makespan function. The next step is choosing the schedule with the lowest value and add the schedule for the second machine by preparing again all possible combinations of a job order. This procedure is repeated for the rest of the machines thus building the solution space tree such as in Figure 15.

The strategy for selecting next nodes in the tree is called *Depth first search* and produces some feasible solution very quickly. All feasible solutions are represented by the leaves in the solution space tree. As soon as the algorithm reaches a leaf and cannot go any deeper, the solution is compared to the best solution found so far and kept in case it is better.

Every time a node is examined its bounds are calculated. In case the node is evaluated (using bounds) as ”non-promising” (does not lead to a better solution), it is fathomed. Otherwise it is processed further as soon as the algorithm reaches it.

The whole solution space tree has exactly $n!$ nodes on every level. Each level represents the number of machines for which all jobs are assigned. Given that, the tree has exactly m levels if we are not counting the root node.

4.1.3.3 Termination criteria are described as certain states of the running algorithm. Algorithm ends in three situations:

- All feasible solutions were examined

Algorithm itself is implemented in such a way that if necessary it examines all possibilities (obviously there is only a finite amount of feasible solutions). Feasible solutions that

are not examined during the processing of the algorithm are those that were fathomed because examining those solution space would only lead to worse results. (Algorithm can fathom only incomplete schedule and such a schedule can only grow in time because we are adding more tasks to the schedule thus extending it.)

- Upper bound equals lower bound

Upper bound and lower bound represent boundaries where the optimal solution that we are looking for is located (indicates the range where makespan of the optimal solution is). Because of the fact that lower bound can only grow and upper bound can only move downwards, when both bounds meet (their values are the same), we have found the optimal solution and cannot improve it anymore. (It might not be the only optimal solution but we cannot find a better one.)

- Time limitation

For high amount of jobs the procedure to find optimal solution can take a very long time. The last external condition for the algorithm is implemented and this condition specifies what the maximum amount of time is, we will let the algorithm running.

Presented termination criteria are valid and will be used in all following implementation not only for simple flow shop scheduling problem.

4.1.3.4 Pseudo code below summarizes the algorithm. The whole implementation is available in the attachment of this thesis.

```
Schedule                # Initial schedule (will be modified in order to get better result)
Number of jobs          # Number of jobs in the Schedule [ 0 ... number of jobs ]
Number of machines      # Number of machines in the Schedule [ 0 ... number of machines ]
Calculated schedule     # Currently processed schedule
Current machine = 0     # Currently processed machine [ 0 ... number of machines ]
Current job = 0         # Currently processed job [ 0 ... number of jobs ]
Available jobs = ∅      # Available jobs represent branches (possible next steps)
Optimal schedule        # Optimal schedule found so far
Upper bound = MAX       # Best calculated solution so far [ 0 ... MAX ]
Lower bound = 0         # Maximal lower bound calculated so far [ 0 ... MAX ]

OptimalSolution(Schedule, Number of machines, Number of jobs, Calculated schedule, Current
    machine, Current job, Available jobs, Best schedule, Upper bound, Lower bound)
 if (Upper bound = Lower bound)
   terminate;                                   # Termination rule
 if (Current job < Number of jobs)              # Not all jobs are scheduled yet
   for each available job in Available jobs     # Get an available job
      Calculated schedule += available job;     # Add job to the schedule
      Available jobs −= available job;          # Remove used job
      OptimalSolution (Schedule, Number of machines, Number of jobs, Calculated schedule,
          Current machine, Current job, Available jobs, Optimal schedule, Upper bound, Lower
          bound)
else
   upper bound = UpperBound(Calculated schedule)    # Calculate upper bound
    if (upper bound < Upper bound)                  # Can I get better bound?
```

```
if (Current machine < Number of machines)          # The schedule is incomplete
   Available jobs = Schedule;                      # Reset list of available jobs
   OptimalSolution (Schedule, Number of machines, Number of jobs, Calculated schedule,
        Current machine, Current job, Available jobs, Optimal schedule, Upper bound, Lower
        bound)
else
   Upper bound = upper bound                       # Update upper bound
   Optimal Schedule = Used;                        # Update best schedule found so far
   lower bound = LowerBound(Calculated schedule)   # Calculate lower bound
   if (lower bound < Lower Bound)                  # Better lower bound was found
      Lower bound = lower bound                    # Update lower bound
```

Statement 3: B&B algorithm for Simple Flow Shop

4.2 Permutation Flow Shop

Compared to a simple flow shop, permutation flow shop scheduling problem definition contains additional condition that job order sequence must be the same on all machines in the schedule. Permutation flow shop is noted as $Fm|perm|C_{max}$. This additional condition limits the amount of all feasible solutions to $n!$, where there are exactly $n!$ ways to create a job sequence.

Like in the simple flow shop problem, the key factors are bounding functions and the method of constructing job sequences in order to get optimal schedule.

Makespan function described in section 4.1.2 works also for permutation flow shop. In B&B for permutation flow shop we are trying to find a schedule that contains the same job sequence on all machines. We will use the makespan function to evaluate incomplete, in the amount of scheduled jobs sense, schedules. Every time we use makespan, we will be using subset of jobs (or all jobs when the schedule is complete) scheduled over all machines.

4.2.1 Lower bound

The lower bound that is used in the implementation was used in [8], where a simple relaxation was shown. If we relax the constrain that each machine can process only one job at a time, for all machines but one, say, M_k, where $k \in (1, \ldots, m)$, then the relaxation can be got by setting *release date* r_j and *delivery time* q_j for all jobs $j \in J$.

$$r_j = \begin{cases} 0 & \text{if } k = 1 \\ \sum_{i=1}^{k-1} p_{ij} & \text{if } k \neq 1 \end{cases}$$

$$q_j = \begin{cases} 0 & \text{if } k = m \\ \sum_{i=k+1}^{m} p_{ij} & \text{if } k \neq m \end{cases}$$

The resulted relaxation is a one-machine problem denoted as $1|r_j, q_j|C_{max}$. We can use this relaxation and prepare a relaxation of $1|r_j, q_j|C_{max}$ by setting release dates and delivery time of all job $j \in J$ to $\min_{j \in J} r_{kj}$ and $\min_{j \in J} q_{kj}$.

	J_1	J_2	J_3
M_1	4	3	1
M_2	3	3	2
M_3	5	4	1

Table 6: Permutation flow shop - Example schedule for lower bound calculation

k = 1		k = 2		k = 3	
$r_{11} = 0$	$q_{11} = p_{21} + p_{31} = 8$	$r_{21} = 4$	$q_{21} = 5$	$r_{31} = p_{11} + p_{21} = 7$	$q_{31} = 0$
$r_{12} = 0$	$q_{12} = p_{22} + p_{32} = 7$	$r_{22} = 3$	$q_{22} = 4$	$r_{32} = p_{12} + p_{22} = 6$	$q_{32} = 0$
$r_{13} = 0$	$q_{13} = p_{23} + p_{33} = 3$	$r_{23} = 1$	$q_{23} = 1$	$r_{33} = p_{13} + p_{23} = 3$	$q_{33} = 0$

Table 7: Permutation flow shop - Example release dates and delivery times for lower bound calculation

Finally the lower bound for a machine M_k, where $k \in (1, \ldots, m)$ is

$$LB_k = \min_{j \in J} r_{kj} + \sum_{j \in J} p_{kj} + \min_{j \in J} q_{kj}$$

and the lower bound for the whole schedule is

$$LB = \max_{1 \leq k \leq m} LB_k.$$

4.2.1.1 Example schedule used for demonstration of lower bound calculation could be given as list processing times like in Table 6. Release dates and delivery times calculation is then shown in Table 7.

Lower bounds for separate machines are:

$LB_1 = p_{11} + p_{12} + p_{13} + q_{13} = 4 + 3 + 1 + 3 = 10$
$LB_2 = r_{23} + p_{21} + p_{22} + p_{23} + q_{23} = 1 + 3 + 3 + 2 + 1 = 10$
$LB_3 = r_{33} + p_{31} + p_{32} + p_{33} = 3 + 5 + 4 + 1 = 13$

Resulted lower bound for the schedule is:

$$LB = max(LB_1, LB_2, LB_3) = max(10, 10, 13) = 13$$

4.2.2 Algorithm

As opposed to implementation for simple flow shop, we are moving only in one dimension - jobs. From a starting point the algorithm is building schedule and examining newly created solution subspace where the optimal solution could be.

4.2.2.1 Initialization At the beginning we have an initial *Schedule* which represents user input. Implementation can calculate optimal solution for only subset of jobs if this Schedule does not contain all jobs.

The algorithm is building up the job sequence using available jobs and *Used* jobs. Jobs cannot be reused in the same schedule therefore there is need to keep track. Anyway, we are starting with an empty schedule so also the list of used jobs is empty.

During the process of dividing the solution subspaces into smaller pieces, these pieces are stored in a set of live nodes called *Branches*. These branches consist of pairs, job sequence (representing already scheduled jobs) and the makespan value (evaluation of this solution subspace). Of course in the beginning there is only whole solution space so this set is empty.

During each moment of the run of the algorithm there is an *Optimal schedule* which holds the current best schedule found so far, which is empty at the starting point, and both bounds. *Upper bound* is set to the potential maximum value and *lower bound* is set to zero.

4.2.2.2 Main idea Starting with an empty schedule, all jobs are available in the beginning. As a first step, the whole solution space is divided into exactly n solution subspaces by simple adding one job from available jobs set to the empty schedule. To evaluate each newly created solution subspace we calculate a makespan of such an incomplete schedule. For the next processing step the incomplete schedule with the smallest value is chosen. In the second step we again decide current solution subspace into even small subspaces by choosing one new available job which we use to extend the current schedule. For all of these newly generated schedules we calculate a makespan and again we choose the schedule with the smallest value for next step. We repeat the previous pattern until all jobs are already used meaning that we have generated a complete schedule.

Every time we reach the leaf in the tree structure (all jobs are scheduled) we calculate also a lower bound and compare our result with the best solution found so far. In case we have found a better solution, it replaces the current best solution.

Once we have a first result a back tracking part of the algorithm starts. We have some yet unexamined solution subspaces represented by an incomplete schedule and their calculated upper bounds. As a next solution subspace to be examined we choose the previously prepared schedule but we are examining it only if its makespan is lower than current upper bound otherwise it is fathomed.

4.2.2.3 Pseudo code shows an algorithm without unnecessary details.

```
Schedule                  # Initial schedule we will try to modify in order to find better result
Used = ∅                  # List of already scheduled jobs
Branches = ∅              # Stores branches for next iterations (set of live nodes)
Optimal Schedule = ∅      # Best schedule found so far
Upper bound = MAX         # Best calculated solution so far [ 0 ... MAX ]
Lower bound = 0           # Maximal lower bound calculated so far [ 0 ... MAX ]

OptimalSolution (Schedule, size(Schedule), Used, size(Used); Optimal Schedule, Upper bound,
    Lower bound)
 if (Upper bound = Lower bound)
   terminate                          # Termination rule
Available = Schedule \ Used           # Prepare set of available jobs
for each Available
   Used = Used + available job        # Add one job to list of used jobs
   upper bound = UpperBound(Used)     # Calculate upper bound
    if (size(Schedule) = size(Used))  # If newly created schedule is complete
      lower bound = LowerBound(Used)  # Calculate lower bound as well
       if (upper bound < Upper Bound) # Better upper bound was found
          Upper Bound = upper bound   # Update upper bound
          Optimal Schedule = Used     # Update best schedule found so far
       if (lower bound < Lower Bound) # Better lower bound was found
          Lower Bound = lower bound   # Update lower bound
   else                               # Schedule is still incomplete
      Branches = (Used, upper bound)  # Add schedule and upper bound into live nodes
   sort(Branches, upper bound)        # Sort branches according to upper bound
   for each Branch
      if (upper bound < Upper Bound)  # Try only if it can lead to better result
         OptimalSolution (Schedule, size(Schedule), Used, size(Used), Optimal Schedule, Upper
             bound, Lower bound)
```

Statement 4: B&B algorithm for Permutation Flow Shop

4.3 No-Wait Flow Shop

Compared to a permutation flow shop scheduling problem the no-wait flow shop contains an additional condition that job must be processed on all machines without delays. No-wait flow shop is noted as $Fm|nwt,(perm)|C_{max}$. It is obvious that feasible schedules for no-wait flow shop are always also permutation schedules.

In the implementation we will use the same algorithm as presented for permutation flow shop but the makespan must be calculated differently. Because of the condition that no delays are allowed, it implies the starting time of a job on a first machine can be delayed. These delays must be taken into consideration during a makespan calculation.

Otherwise, the procedure about generating a solution space tree, lower bound calculation, termination criteria and even the strategy for choosing next solution subspace for examination can be used as presented before in section about permutation flow shop.

4.3.1 Makespan

For makespan calculation function we will use two facts.

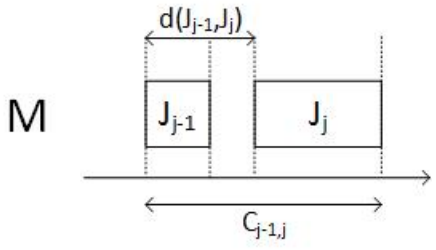


Figure 16: No-wait Flow Shop - Calculating processing delays

	J_1	J_2	J_3
M_1	1	8	2
M_2	5	3	4
M_3	2	2	9

Table 8: No-wait flow shop - Example assignment for makespan calculation

Definition 4.1 *Makespan for* $F2|nwt,(perm)|C_{max}$ *and* $F2|perm|C_{max}$ *is the same [9].*

Definition 4.2 *In the no-wait flow shop scheduling problem the difference between the completion time of a job's last task and the starting time of the first operation is equal to the sum of its operation times on all machines [9].*

Let $d(J_{j-1}, J_j)$ represents a time period between the start of processing $T_{1(j-1)}$ and the completion of T_{mj}.

$$d(J_{j-1}, J_j) = C_{(j-1)j} - \sum_{i=1}^{m} p_{ij},$$

where $j = 2, 3, \ldots, n$ and $C_{(j-1)j}$ represents completion time of a schedule consisting of exactly two jobs J_{j-1} and J_j.

Makespan of the schedule is then calculated as follows:

$$C_{max} = \sum_{j=2}^{n} d(J_{j-1}, J_j) + \sum_{i=1}^{m} p_{in}$$

4.3.1.1 Example assigment for calculating makespan can be $F3|nwt,(perm)|C_{max}$, where processing times are listed in the Table 8.

Steps leading to the makespan value for a given schedule would than be:

$$d(1,2) = C_{1,2}(J_2, 3) - \sum_{i=1}^{3} p_{i2} = 14 - 13 = 1$$

$$d(2,3) = C_{2,3}(J_3, 3) - \sum_{i=1}^{3} p_{i3} = 24 - 15 = 9$$

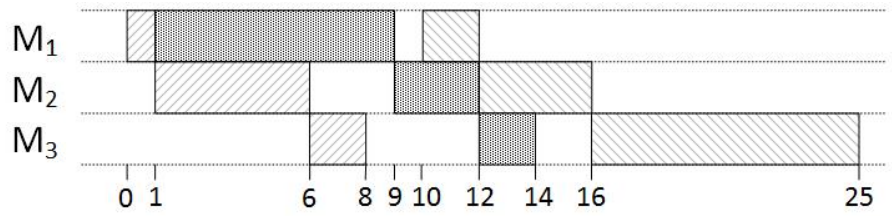


Figure 17: No-wait Flow Shop - Makespan example Gantt chart

$$C_{max} = d(1,2) + d(2,3) + \sum_{i=1}^{3} p_{i3} = 1 + 9 + 15 = 25$$

Correspoding visualization in Figure 17 shows that the calculated value is correct.

4.4 Blocking Flow Shop

Blocking flow shop is not very different from no-wait flow shop. It is even the same but only in case of two machines, where $F2|nwt, (perm)|C_{max} = F2|block, (perm)|C_{max}$. For more than two machines we can see the difference when we have to take into account that there are blocking sections. These sections represent the time period when a completed task of a job J_i is blocking machine M_i and it cannot start processing next task on machine M_{i+1}. These sections and their parameters are important in the calculation of makespan and lower bound.

4.4.1 Makespan

Before we begin with makespan calculation we need to extend schedule parameters by a departure time. Let d_{ij} be a departure time of task T_{ij}, where, as usual, i represents i^{th} machine and j represents j^{th} job. Departure time d_{ij} can be calculated as follows [10]:

$$d_{01} = 0,$$

$$d_{i1} = \sum_{q=1}^{i} p_{q1},$$

where $i = 1, \ldots, m-1$,

$$d_{0j} = d_{1(j-1)},$$

where $j = 2, \ldots, n$,

$$d_{ij} = max\{d_{(i-1)j} + p_{ij}, d_{(i+1)(j-1)}\},$$

where $i = 1, \ldots, m-1$ and $j = 2, \ldots, n$,

$$d_{mj} = d_{(m-1)j} + p_{mj},$$

	J_1	J_2	J_3
M_1	1	8	4
M_2	5	3	2
M_3	2	2	9

Table 9: Blocking flow shop - Example assignment for makespan calculation

d_{33}	=	$d_{23} + p_{33}$	=	24
d_{23}	=	MAX($d_{13} + p_{23}$; d_{32})	=	15
d_{13}	=	MAX($d_{03} + p_{13}$; d_{22})	=	13
d_{32}	=	$d_{22} + p_{32}$	=	14
d_{03}	=	d_{12}	=	9
d_{22}	=	MAX($d_{12} + p_{22}$; d_{31})	=	12
d_{12}	=	MAX($d_{02} + p_{12}$; d_{21})	=	9
d_{31}	=	$d_{21} + p_{31}$	=	8
d_{02}	=	d_{11}	=	1
d_{21}	=	$p_{11} + p_{21}$	=	6
d_{21}	=	p_{11}	=	1

Table 10: Blocking flow shop - Example makespan calculation steps

where $j = 1, \ldots, n$.

From the above-mentioned you can see that d_{0j}, where $j = 1, \ldots, n$, denotes the starting time of job J_j on the first machine. d_{mj} represents the completion time of job J_j in the schedule. It is obvious that the departure times must be calculated recursively but in the end we get makespan as

$$C_{max} = max_{j=1,\ldots,n} d_{mj} = d_{mn}.$$

4.4.1.1 Example schedule and its calculation goes recursively from end to the beginning and once it reaches the beginning which is defined, then it adds more and more till the end of the calculation.

In the example we are looking for a departure time of the the job on a last machine given Table 9.

In the Tabel 10 you can see the calculation process of a makespan. Each step is showing how the recursion goes down. Two bottom lines can be calculated based just on processing times. Having last two lines completed we can move back upwards and calculate the rest step by step. At the end we have d_{33} which represents the departure time of the last task of the last job in the schedule.

We can check the calculated value in the Gantt chart visualisation of the schedule in Figure 18.

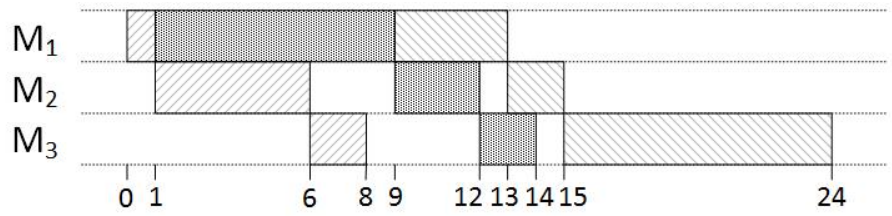


Figure 18: Blocking Flow Shop - Makespan example Gantt chart

4.4.2 Lower bound

Using the already presented method for calculating makespan we can prepare a lower bound calculation. The following method was presented in [11] together with an explanation about relation to a directed graph representation.

First of all, let's PS contain partial job sequence, only assigned jobs. And NPS containing all unscheduled jobs.

$$PS = (1, 2, \ldots, s),$$
$$NPS = (s+1, s+2, \ldots, n)$$

Cardinality of $|PS|$ indicates how many jobs are already assigned in the schedule and of course how many are not. The lower bound for a machine M_i can be calculated as follows:

$$LB(i) = max(d_{i|PS|} + p_{i(|PS|+1)}, d_{(i+1)|PS|}) + \sum_{r=|PS|+1}^{n-1} max(p_{i(r+1)}, p_{(i+1)r}) + \sum_{q=i+1}^{m} p_{qn}$$

where $p_{(m+1)r} = 0$ and $d_{(m+1)r} = 0$ for all r.

For a $Fm|block, (perm)|C_{max}$ with a given partial schedule a lower bound on the makespan is

$$LB = min_{1 \leq i \leq m} LB(i).$$

4.4.2.1 Example from the previous section where only first job J_1 is assigned and jobs J_2 and J_3 are unassigned has lower bound calculated as follows:

$LB(1) = max(d_{11}+p_{12}, d_{21})+max(p_{13}, p_{22})+p_{23}+p_{33} = max(6,9)+max(4,3)+11 = 24$
$LB(2) = max(d_{21} + p_{22}, d_{31}) + max(p_{23}, p_{32}) + p_{33} = max(9,8) + max(2,2) + 9 = 20$
$LB(3) = max(d_{31} + p_{32}, d_{41}) + max(p_{33}, p_{42}) = max(10,0) + max(9,0) = 19$

$LB = min(LB(1), LB(2), LB(3)) = min(24, 20, 19) = 19$

Calculated lower bound will server as a evaluation of a solution subspace where only one job is assigned.

4.4.3 Algorithm

For the blocking flow shop, a slightly modified version of algorithm for permutation flow shop will be used. This time, schedule is evaluated by lower bound function. In case a solution subspace is evaluated with a value which is higher then makespan of the best schedule found so far, it is fathomed. Makespan function is used only on complete schedules and calculated upper bound states the completion time of the current best schedule.

```
Schedule                  # Initial schedule we will try to modify in order to find better result
Used = ∅                  # List of already scheduled jobs
Branches = ∅              # Stores branches for next iterations (set of live nodes)
Optimal Schedule = ∅      # Best schedule found so far
Upper bound = MAX         # Best calculated solution so far [ 0 ... MAX ]
Lower bound = MAX         # Minimal lower bound calculated so far [ 0 ... MAX ]

OptimalSolution (Schedule, size(Schedule), Used, size(Used); Optimal Schedule, Upper bound,
      Lower bound)
 if (Upper bound = Lower bound)
    terminate                                  # Termination rule
Available = Schedule \ Used                    # Prepare set of available jobs
for each Available
    Used = Used + available job                # Add one job to list of used jobs
    if (size(Schedule) = size(Used))           # If newly created schedule is complete
       upper bound = UpperBound(Used)          # Calculate upper bound
       if (upper bound < Upper Bound)          # Better upper bound was found
          Upper Bound = upper bound            # Update upper bound
          Optimal Schedule = Used              # Update best schedule found so far
    else                                       # The schedule is still incomplete
       lower bound = LowerBound(Used)          # Calculate lower bound
       if (lower bound < Lower Bound)          # Better lower bound was found
             Lower Bound = lower bound         # Update lower bound
       if (lower bound < Upper Bound)          # Better solution can be find
             Branches = (Used, lower bound)    # Add schedule and lower bound into live nodes
sort(Branches, lower bound)                    # Sort branches according to upper bound
for each Branch
    OptimalSolution (Schedule, size(Schedule), Used, size(Used), Optimal Schedule, Upper bound,
         Lower bound)
```

Statement 5: B&B algorithm for Blocking Flow Shop

5 Computational results

To verify, demonstrate and calculate partial results that are the expected outcome of the thesis, we have a set of flow shop problem instances. For each instance the algorithm runs on a server with 32 Intel Xeon 2 GHz processors with the total amount of 64 GB RAM. The algorithm for one instance can only use one processor and also memory requirements are negligible, because of that multiple instances could have been executed without any interference between each run.

Taillard suite of problem instances originally prepared for $Fm||C_{max}$, where instance sizes range from (5,20) (i.e. 5 machines and 20 jobs) to (50,1000), is used as a set of benchmark instances for calculations. The complete collection of problem instance and calculated results can be found in the attachment.

The follwoing table represents just a portion of results but it should be enough to demonstrate the trends that can be observed in all results. The Table 11 contains an instance size, reference to a specific problem instance and bounds for each flow shop problem variation.

5.1 Observations

The least amount of partial results were generated by the implementation for *blocking flow shop*, where the most complex lower bound was used. This together with a depth first search strategy does not seem to be a good choice because even schedule containing 20 job and 10 machines was too difficult for calculation.

Lower bound function always uses both assigned and also unassigned jobs. It means that no matter how many jobs are assigned the calculation takes approximatelly the same amount of time.

The simple flow shop contains the highest number of feasible solution because of missing condition that only permutation schedules are allowed. On the other hand the lower bound function is very simple and can be easily calculated. The results are showing that calculated lower bounds for simple flow shop for all instances gives the lowest values.

Very interesting situation can be see when comparing upper bounds for simple flow shop and permutation flow shop. As you can see in the Table 11 the upper bound values for permutation flow shop are better then for simple flow shop. It can be seen that not following the same job order on all machines leads often to worse results.

The best upper bounds were calculated for Permutation flow shop. This is because the resulting schedule does not contain any delays or waiting tiem periods.

		FSS		FSP		FSNoWait		FSBlock	
$n \times m$	Instance	C_{max}	LB	C_{max}	LB	C_{max}	LB	C_{max}	LB
20×5	ta001	1448	1121	1342	1232	1607	1232	1531	1110
	ta002	1545	1207	1501	1290	1729	1290	1664	1290
	ta003	1597	1000	1312	1073	1597	1073	1613	1073
	ta004	1754	1177	1507	1268	1788	1268	1718	1265
	ta005	1431	1107	1305	1198	1657	1198	1525	1149
	ta006	1616	1122	1412	1180	1626	1180	1564	1120
	ta007	1528	1152	1395	1226	1652	1226	1676	1148
	ta008	1428	1097	1370	1170	1697	1170	1596	1148
	ta009	1468	1138	1396	1206	1633	1206	1572	1206
	ta0010	1404	1009	1313	1082	1498	1082	1512	1059

Table 11: Calculation results

6 Conclusion

The simple flow shop scheduling problem is computationally the most difficult one from the presented scheduling problems. The amount of feasible solutions can be very big which happened to lead to show the implementation limitations where it cannot be calculated without a special handling in the code.

The chosen strategy - depth first search - greatly impacts the outcomes of the calculations because the chosen branch of the solution space tree greatly determines the examined part of the tree. It can be seen that first chosen branch is examined in detail while it very rarely happens that the calculation jumps to a different branch depending on the depth of the current and branch resp. the amount of feasible solution this branch contains.

B&B algorithm with a depth first search strategy seems to be a good candidate from implementation using parallelism. The idea, where multiple branches are simultaneously processed with a shared location, where current best solution together with both bounds is stored, suggests itself.

For further study I would recommend a comparison between different node selection strategies for a single problem definition. The results should give better understanding about which strategy is the most suitable for flow shop scheduling problems. The node selection strategy is the key part of the resulting algorithm.

The calculated results presented in previous chapter show that more restricted flow shop problem definition gives the better foundation for makespan and lower bound functions which in the end leads to better results (tighter bounds). The results that were gathered will serve as a basis for next examination by evolutionary algorithms.

Bc. Pavel Ivánek

7 References

[1] LAND, A. H. and DOIG, A. G. *An automatic method for solving discrete programming problems.* Econometrica, July 1960, vol. 28, no. 3., pp. 427-520.

[2] CLAUSEN, Jens, *Branch and Bound Algorithms - Principles and Examples*. Denmark, 1999. Department of Computer Science, University of Copenhagen.

[3] PINEDO, Michael L. *Scheduling: Theory, Algorithms, and Systems*. Fourth edition. New York: Springer-Verlag New York, 2012. ISBN: 978-4-4614-1986-0.

[4] EMMONS, E. - VAIRAKTARAKIS, G. Flow Shop Scheduling: Theoretical Results, Algoriths, and Applications. *Operations Research and Management Science*, vol. 182. New York: Springer Science+Business Media New York, 2013. ISBN: 978-1-4614-5152-5.

[5] GRAHAM, R. L. - LAWLER, E. L. - LENSTRA, J. K. - RINNOOY KAN, A. H. G. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 1979, vol. 5., pp. 287-326.

[6] YEUNG, W.-K. - OGUZ, C. - CHENG, T.-C. E. Two-machine Flow Shop Scheduling with a Common Due Window to Minimize Earliness and Tardiness. *Naval Research Logistics*, October 2009, vol. 56, issue 7, pp. 593-599.

[7] JOHNSON, S. M. Optimal Two- and Three-Stage Production Schedules with Setup Times Included. *Naval Research Logistics*, November 1954, vol. 1, pp. 55-60.

[8] LADHARI, Talel and HAOUARI, Mohamed. A computational study of the permutation flow shop problem based on tight lower bound. *Computer & Operations Research*, July 2005, vol. 32, issue 7, pp. 1831-1847.

[9] GAO, Kaizhou - XIE, Shengxian - JIANG, Hua - LI, Junqing. *Discrete Harmony Search Algorithm for the No Wait Flow Shop Scheduling Problem with Makespan Criterion*. Berlin: Springer Berlin Heidelberg, 2012, p.592-599. ISBN: 978-3-642-24727-9.

[10] PAN, Quan-Ke - WANG Ling. Effective heuristics for the blocking flow shop scheduling problem with makespan minimization. *Omega*, April 2012 , vol. 40, issue 2, pp. 218-229.

[11] RONCORI, D. P. A Branch-and-Bound Algorithm to Minimize the Makespan in a Flowshop with Blocking. *Anals of Operations Research*, September 2005, vol. 138, issue 1, pp. 53-65. ISSN: 0254-5330.

A Complete set of results

The following set of tables presents the complete list of results. Tables contains dataset instances used as inputs, size of instances and calculated bounds for all four flow shop variations presented in the thesis. The best schedules that were found during the calculations can be found on the attached DVD.

Example file with output messages:

Calculating optimal solution for Permutation Flow Shop problem using Branch and Bound algorithm.
Current lower bound: 1219
Current lower bound: 1230
Current upper bound: 1531
Current best schedule:
3 17 9 16 1 12 8 19 13 14 15 7 20 11 2 6 4 10 5 18

		FSS		FSP		FSNoWait		FSBlock	
$n \times m$	Instance	C_{max}	LB	C_{max}	LB	C_{max}	LB	C_{max}	LB
20×5	ta001	1448	1121	1342	1232	1607	1232	1531	1110
	ta002	1545	1207	1501	1290	1729	1290	1664	1290
	ta003	1597	1000	1312	1073	1597	1073	1613	1073
	ta004	1754	1177	1507	1268	1788	1268	1718	1265
	ta005	1431	1107	1305	1198	1657	1198	1525	1149
	ta006	1616	1122	1412	1180	1626	1180	1564	1120
	ta007	1528	1152	1395	1226	1652	1226	1676	1148
	ta008	1428	1097	1370	1170	1697	1170	1596	1148
	ta009	1468	1138	1396	1206	1633	1206	1572	1206
	ta010	1404	1009	1313	1082	1498	1082	1512	1059
20×10	ta011	2004	1178	1852	1448	2421	1448	n/a	1369
	ta012	2104	1177	1969	1479	2439	1479	n/a	1474
	ta013	1812	1217	1661	1407	2194	1407	n/a	1172
	ta014	1726	1071	1702	1308	2056	1308	n/a	1151
	ta015	1944	1154	1649	1325	2225	1325	n/a	1224
	ta016	1877	1099	1596	1290	2062	1290	n/a	1102
	ta017	1935	1194	1577	1388	2202	1388	n/a	1221
	ta018	2044	1108	1640	1363	2344	1363	n/a	1135
	ta019	1978	1251	1710	1472	2078	1472	n/a	1334
	ta020	2051	1158	1782	1356	2421	1356	n/a	1238
20×20	ta021	2770	1217	2414	1911	3474	1911	n/a	1683
	ta022	2543	1084	2547	1711	3059	1711	n/a	1634
	ta023	2625	1159	2508	1844	3328	1844	n/a	1795
	ta024	2800	1226	2624	1810	3336	1810	n/a	1662
	ta025	2829	1188	2581	1899	3268	1899	n/a	1661
	ta026	2597	1218	2476	1875	3161	1875	n/a	1714
	ta027	2723	1175	2531	1875	3311	1875	n/a	1866
	ta028	2697	1288	2436	1880	3378	1880	n/a	1635
	ta029	2713	1183	2583	1840	3517	1840	n/a	1747
	ta030	2830	1223	2479	1900	3186	1900	n/a	1740
50×5	ta031	3095	2674	3323	2712	3641	2712	n/a	2282
	ta032	3515	2742	3641	2808	3918	2808	n/a	2808
	ta033	2900	2551	3417	2596	3777	2596	n/a	2493
	ta034	3073	2679	3562	2740	4000	2740	n/a	2740
	ta035	3071	2784	3626	2837	3969	2837	n/a	2348
	ta036	3195	2744	3190	2793	3877	2793	n/a	2726
	ta037	3450	2625	3307	2689	3718	2689	n/a	2655
	ta038	3140	2650	3376	2667	3791	2667	n/a	2563
	ta039	2930	2469	3009	2527	3502	2527	n/a	2135
	ta040	3188	2718	3705	2776	3976	2776	n/a	2677

Table 12: Complete calculation results - part 1

		FSS		FSP		FSNoWait		FSBlock	
$n \times m$	Instance	C_{max}	LB	C_{max}	LB	C_{max}	LB	C_{max}	LB
50×10	ta041	3754	2730	3902	2907	4937	2907	n/a	2669
	ta042	3685	2538	4014	2821	4906	2821	n/a	2821
	ta043	3612	2683	4009	2801	4970	2801	n/a	2695
	ta044	3669	2784	4047	2968	5035	2968	n/a	2496
	ta045	3741	2718	4066	2908	5149	2908	n/a	2851
	ta046	3736	2694	4073	2941	4978	2941	n/a	2761
	ta047	3678	2779	4066	3062	5163	3062	n/a	2623
	ta048	3773	2780	3929	2959	4909	2959	n/a	2814
	ta049	3792	2653	3944	2795	5152	2795	n/a	2095
	ta050	3845	2763	4052	3046	4951	3046	n/a	2972
50×20	ta051	5094	2897	4955	3480	7165	3480	n/a	3224
	ta052	4730	2894	4705	3424	6724	3424	n/a	3329
	ta053	4592	2751	4782	3351	6723	3351	n/a	3351
	ta054	4797	2800	4828	3336	6701	3336	n/a	3229
	ta055	4748	2793	5052	3313	7265	3313	n/a	3276
	ta056	4946	2918	4776	3460	6821	3460	n/a	2881
	ta057	4742	2876	4961	3427	6873	3427	n/a	3076
	ta058	4763	2813	4998	3383	6771	3383	n/a	3359
	ta059	4823	2912	4709	3457	6913	3457	n/a	2978
	ta060	4901	2829	5127	3438	6893	3438	n/a	2996
100×5	ta061	5943	5381	6808	5437	7498	5437	n/a	5301
	ta062	5878	5162	6744	5208	7263	5208	n/a	5208
	ta063	5880	5108	6605	5130	7349	5130	n/a	5130
	ta064	5675	4935	6237	4963	6931	4963	n/a	4874
	ta065	6095	5174	6551	5195	7310	5195	n/a	5190
	ta066	5753	5032	6675	5063	7496	5063	n/a	5063
	ta067	5935	5181	6562	5198	7401	5198	n/a	4693
	ta068	6068	5003	6679	5038	7418	5038	n/a	5038
	ta069	6193	5363	6898	5385	7395	5385	n/a	5385
	ta070	6157	5254	6808	5272	7529	5272	n/a	5069
100×10	ta071	6983	5636	7670	5759	9373	5759	n/a	5217
	ta072	6558	5182	7370	5345	9260	5345	n/a	5103
	ta073	6667	5500	7474	5623	9273	5623	n/a	4676
	ta074	7300	5539	7695	5732	9660	5732	n/a	5624
	ta075	6844	5290	7396	5431	9357	5431	n/a	4847
	ta076	6591	5114	7520	5246	9220	5246	n/a	5131
	ta077	6765	5401	7327	5523	9304	5523	n/a	4642
	ta078	6517	5333	7585	5556	9250	5556	n/a	5449
	ta079	6859	5566	7626	5779	9505	5779	n/a	5779
	ta080	6930	5576	7390	5830	9465	5830	n/a	5830

Table 13: Complete calculation results - part 2

		FSS		FSP		FSNoWait		FSBlock	
$n \times m$	Instance	C_{max}	LB	C_{max}	LB	C_{max}	LB	C_{max}	LB
100×20	ta081	7854	5357	8461	5851	12749	5851	n/a	5617
	ta082	7591	5500	8646	6099	12104	6099	n/a	5184
	ta083	7755	5536	8387	6099	12077	6099	n/a	5521
	ta084	7885	5614	8717	6072	12702	6072	n/a	5610
	ta085	7729	5445	8509	6009	12199	6009	n/a	5433
	ta086	8072	5584	8480	6144	12197	6144	n/a	6144
	ta087	8033	5507	8556	5991	12824	5991	n/a	4995
	ta088	8138	5472	8786	6084	12539	6084	n/a	5907
	ta089	7907	5395	8496	5979	12310	5979	n/a	5642
	ta090	8099	5772	8712	6298	12624	6298	n/a	5162
200×10	ta091	12193	10616	13984	10816	17602	10816	n/a	9790
	ta092	12796	10230	13928	10422	17219	10422	n/a	10245
	ta093	12556	10699	14101	10886	17583	10886	n/a	10886
	ta094	12198	10647	14283	10794	17400	10794	n/a	10794
	ta095	12110	10313	14413	10437	17474	10437	n/a	10366
	ta096	12116	10124	13739	10255	17596	10255	n/a	9986
	ta097	12848	10612	14506	10761	17631	10761	n/a	10761
	ta098	12294	10480	14193	10663	17605	10663	n/a	10250
	ta099	12010	10259	13872	10348	17483	10348	n/a	10133
	ta100	12274	10451	14391	10616	17532	10616	n/a	10616
200×20	ta101	13576	10498	14945	10979	22557	10979	n/a	10054
	ta102	13628	10445	15545	10947	23253	10947	n/a	10883
	ta103	14152	10642	15385	11150	22525	11150	n/a	10392
	ta104	13479	10616	15361	11127	22696	11127	n/a	10242
	ta105	13686	10674	15394	11132	22925	11132	n/a	10673
	ta106	13917	10653	15493	11085	23242	11085	n/a	10424
	ta107	13836	10764	15539	11194	23089	11194	n/a	10736
	ta108	13855	10597	15442	11126	22817	11126	n/a	10326
	ta109	13409	10384	14762	10965	23012	10965	n/a	10321
	ta110	14101	10607	15360	11122	22489	11122	n/a	10446
500×20	ta111	30121	25464	35302	25922	52376	25922	n/a	24553
	ta112	31202	25898	35985	26353	53603	26353	n/a	26086
	ta113	30447	25789	35230	26320	52278	26320	n/a	24575
	ta114	30355	25903	35499	26424	52957	26424	n/a	25965
	ta115	30099	25693	35891	26181	52686	26181	n/a	26143
	ta116	30946	25841	35734	26401	53063	26401	n/a	25561
	ta117	30792	25793	35687	26300	52564	26300	n/a	25244
	ta118	31034	26012	35428	26429	53190	26429	n/a	25807
	ta119	30634	25405	36227	25891	52480	25891	n/a	25468
	ta120	30148	25861	35451	26315	52638	26315	n/a	25107

Table 14: Complete calculation results - part 3

		FSS		FSP		FSNoWait		FSBlock	
$n \times m$	Instance	C_{max}	LB	C_{max}	LB	C_{max}	LB	C_{max}	LB
500×50	ta121	35150	26269	38217	27833	77060	27833	n/a	26170
	ta122	35089	26469	39144	28209	76703	28209	n/a	26925
	ta123	35190	26318	39254	28095	77022	28095	n/a	26590
	ta124	35143	26405	38825	27997	77010	27997	n/a	27408
	ta125	35020	26697	39512	28325	78006	28325	n/a	27457
	ta126	35238	26650	39429	28266	78162	28266	n/a	26928
	ta127	35309	26726	39218	28387	76613	28387	n/a	27143
	ta128	35255	26237	39162	27940	77136	27940	n/a	26917
	ta129	35313	26296	39005	27797	76799	27797	n/a	26642
	ta130	34938	26389	39027	28114	76209	28114	n/a	27099
700×20	ta131	40717	36254	49498	36720	72283	36720	n/a	34721
	ta132	41095	35932	49604	36391	72003	36391	n/a	34467
	ta133	41351	36190	48959	36671	71698	36671	n/a	35725
	ta134	42016	36480	49200	36844	71862	36844	n/a	34501
	ta135	41416	36689	49494	37090	71844	37090	n/a	35016
	ta136	40986	36162	49038	36625	71745	36625	n/a	35042
	ta137	41526	36409	48576	36954	72708	36954	n/a	35144
	ta138	41963	36907	49036	37297	71979	37297	n/a	35392
	ta139	41383	36032	49050	36544	71797	36544	n/a	35371
	ta140	41664	37089	49034	37661	72003	37661	n/a	35622
700×50	ta141	n/a	n/a	53062	38793	104439	38793	n/a	36796
	ta142	n/a	n/a	53173	38568	105316	38568	n/a	37889
	ta143	n/a	n/a	53323	38144	104034	38144	n/a	37793
	ta144	n/a	n/a	52925	37516	104761	37516	n/a	35918
	ta145	n/a	n/a	52906	38374	105063	38374	n/a	37866
	ta146	n/a	n/a	52681	38128	105165	38128	n/a	37636
	ta147	n/a	n/a	53100	38864	104814	38864	n/a	37405
	ta148	n/a	n/a	52507	37438	103917	37438	n/a	36753
	ta149	n/a	n/a	52285	38203	104559	38203	n/a	36599
	ta150	n/a	n/a	53099	39076	105535	39076	n/a	38527
1000×20	ta151	58117	51566	69413	52081	100532	52081	n/a	51485
	ta152	58035	51722	68731	52130	100515	52130	n/a	49290
	ta153	57079	51484	68356	51878	100483	51878	n/a	48557
	ta154	56155	51297	68817	51831	99989	51831	n/a	50283
	ta155	57217	51567	68943	52134	99936	52134	n/a	50626
	ta156	57705	52028	69708	52482	100350	52482	n/a	50660
	ta157	56712	50968	70092	51400	100032	51400	n/a	51301
	ta158	58786	51169	69335	51734	100601	51734	n/a	51386
	ta159	57053	50801	68436	51252	99809	51252	n/a	50367
	ta160	57319	51451	70089	51897	100415	51897	n/a	51333

Table 15: Complete calculation results - part 4

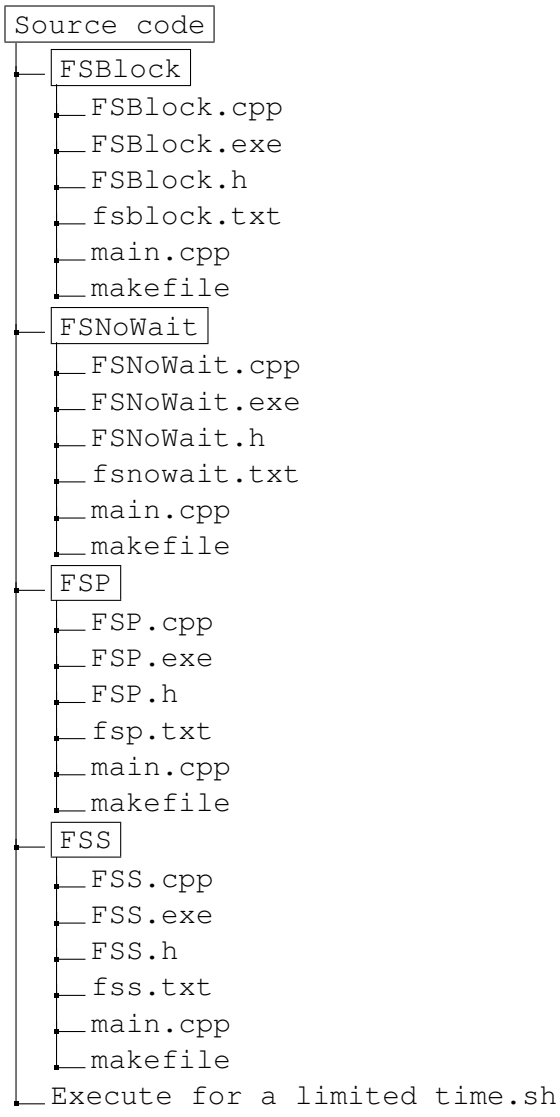
		FSS		FSP		FSNoWait		FSBlock	
$n \times m$	Instance	C_{max}	LB	C_{max}	LB	C_{max}	LB	C_{max}	LB
1000×50	ta161	n/a	n/a	72574	53932	145656	53932	n/a	50733
	ta162	n/a	n/a	73682	53949	145336	53949	n/a	52010
	ta163	n/a	n/a	73723	52804	145725	52804	n/a	51589
	ta164	n/a	n/a	73768	54132	145655	54132	n/a	52729
	ta165	n/a	n/a	73857	54281	145341	54281	n/a	53359
	ta166	n/a	n/a	72793	53094	145383	53094	n/a	51294
	ta167	n/a	n/a	73602	54584	144540	54584	n/a	53225
	ta168	n/a	n/a	73130	53429	146038	53429	n/a	50871
	ta169	n/a	n/a	72824	53481	145125	53481	n/a	52132
	ta170	n/a	n/a	73028	53717	144636	53717	n/a	52425

Table 16: Complete calculation results - part 5

B Annex on DVD

Attached DVD contais number of files containing implementation. There are four separate implementations for each flow shop variation - No-wait, Blocking, Simple and Permutation. Each implementation folder contains the same list of files. Implementation files for simple flow shop are described in Table 17. Binary file must be executed with one parameter - input file, e.g. *FSBlock.exe fsblock.txt*.

Last file *Execute for a limited time.sh* contains a bash script that is used for performance testing. This sript contains the time limitation in case the execution takes too long.



File Name	Description
FSS.cpp	Source file containing implementation of class specific for the problem
FSS.h	Header file
fss.txt	Sample input
main.cpp	Main file containing predefined routine for demonstration purposes
makefile	Makefile

Table 17: Description of files with source code

Another separate folder contains datasets and calculated results containing lower bound(s), upper bound(s) and also best schedule found.

```
Data
├── ta001.txt
├── ta001.txt_FSBlock
├── ta001.txt_FSNoWait
├── ta001.txt_FSP
├── ta001.txt_FSS
├── ta002.txt
├── ta002.txt_FSBlock
├── ta002.txt_FSNoWait
├── ta002.txt_FSP
├── ta002.txt_FSS
├── ...
├── ...
├── ...
├── ...
├── ...
├── ta170.txt
├── ta170.txt_FSBlock
├── ta170.txt_FSNoWait
├── ta170.txt_FSP
└── ta170.txt_FSS
```